Swift设计模式(iOS)

Yourtion

Published with GitBook



目錄

介紹	0
iOS 设计模式	1
入门 - 开始	2
设计模式之王 - MVC	3
如何使用 MVC 模式	3.1
单例模式 - Singleton	4
如何使用单例模式	4.1
外观模式 - Facade	5
如何使用外观模式	5.1
装饰者模式 - Decorator	6
扩展	6.1
如何使用扩展	6.2
委托	6.3
如何使用委托模式	6.4
适配器模式 - Adapter	7
如何使用适配器模式	7.1
观察者模式 - Observer	8
通知 - Notification	8.1
键值观察 - KVO	8.2
备忘录模式 - Memento	9
如何使用备忘录模式	9.1
归档 - Archiving	9.2
最后的润色	10
入门 - 小结	11

Swift设计模式 (iOS)

我们将会通过完成一个完整的应用，展示音乐专辑和专辑的相关信息来学习设计模式在 Swift 中的实现。

通过这个应用，我们会接触一些 Cocoa 中常见的设计模式：

- 创建型 (Creational)：单例模式 (Singleton)
- 结构型 (Structural)：MVC、装饰者模式 (Decorator)、适配器模式 (Adapter)、外观模式 (Facade)
- 行为型 (Behavioral)：观察者模式 (Observer)、备忘录模式 (Memento)

整理排版说明

在 `Xcode 7` 中进行编码测试，升级为 `Swift 2.0` 解决原文中出现的问题，保证了语句与Demo的可用性

在线阅读: http://swift-design-patterns.books.yourtion.com/

下载电子书: https://www.gitbook.com/book/yourtion/swiftdesignpatterns/details

直接下载：PDF、EPub、Mobi

有修改建议优化请提交**Issus**，或请直接**Fork**：**https://github.com/yourtion/SwiftDesignPatterns/** 进行修改并申请 **Pull Request**。

项目Demo：https://github.com/yourtion/SwiftDesignPatterns-Demo1

更新声明

本书整理排版自：

- iOS 中的设计模式 (Swift版本) Part 1
- iOS 中的设计模式 (Swift版本) Part 2

原文翻译自 Introducing iOS Design Patterns in Swift – Part 1/2 和 Introducing iOS Design Patterns in Swift – Part 2/2 ， 本教程 objc 版本的作者是 Eli Ganem ， 由 Vincent Ngo 更新为 Swift 版本。

Update 04/22/2015: Updated for Xcode 6.3 and Swift 1.2.

Update note: This tutorial was updated for iOS 8 and Swift by Vincent Ngo. Original post by Tutorial team member Eli Ganem.

GitBook 排版

Yourtion

- yourtion@gmail.com
- https://github.com/yourtion

iOS 设计模式

说到设计模式，相信大家都不陌生，但是又有多少人知道它背后的真正含义？绝大多数程序员都知道设计模式十分重要，不过关于这个话题的文章却不是很多，开发者们在开发的时候有时也不太在意设计模式方面的内容。

设计模式针对软件设计中的常见问题，提供了一些可复用的解决方案，开发者可以通过这些模板写出易于理解且能够复用的代码。正确的使用设计模式可以降低代码之间的耦合度，从而很轻松的修改或者替换以前的代码。

如果你对设计模式还很陌生，那么告诉你一个好消息！在 iOS 的开发过程中，其实你不知不觉已经用了很多设计模式。这得益于 Cocoa 提供的框架和一些良好的编程习惯。接下来的这篇教程将会带你一起飞，去领略设计模式的魅力。

常见模式

第一部分我们将会完成一个完整的应用，展示音乐专辑和专辑的相关信息。

通过这个应用，我们会接触一些 Cocoa 中常见的设计模式：

- 创建型 (Creational)：单例模式 (Singleton)
- 结构型 (Structural)：MVC、装饰者模式 (Decorator)、适配器模式 (Adapter)、外观模式 (Facade)
- 行为型 (Behavioral)：观察者模式 (Observer)、备忘录模式 (Memento)

嘿嘿嘿别愁眉苦脸的嘛，这篇文章不是什么长篇大论的理论知识，你会在开发应用的过程中慢慢学会这些设计模式。

先来预览一下最终的结果：

Carrier 9:51 AM

Pop Music

Sting

U2

STARING AT THE SUN

Artist David Bowie

Album Best of Bowie

Genre Pop

Year 1992

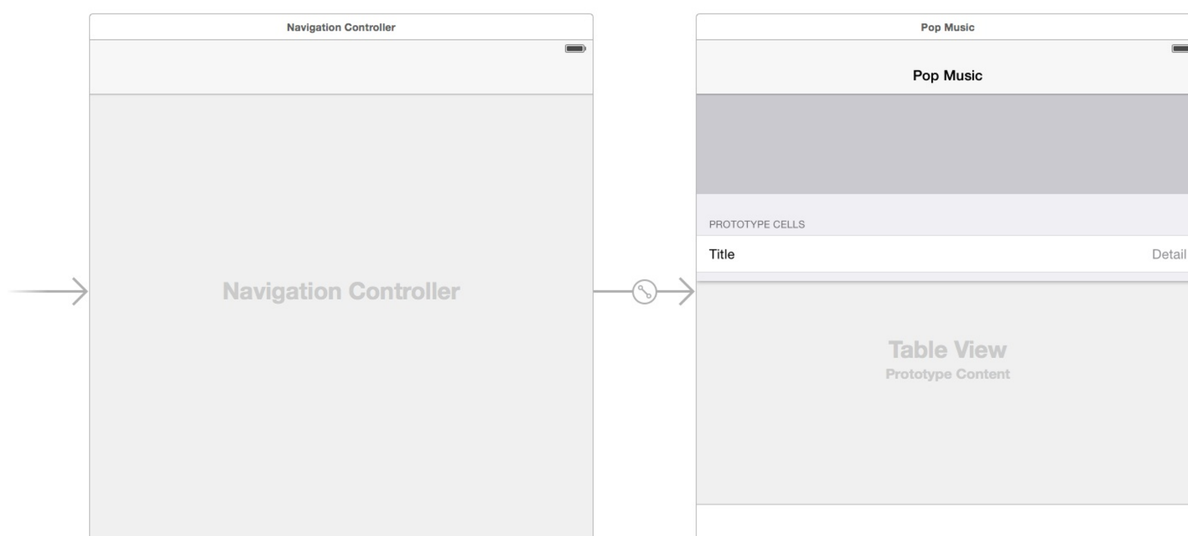
Undo

开始

下载初始项目并解压，在 Xcode 中打开 `BlueLibrarySwift.xcodeproj` 项目文件。

项目中有三个地方需要注意一下：

- `ViewController` 有两个 `IBOutlet` ，分别连接到了 `UITableView` 和 `UIToolBar` 上。
- 在 StoryBoard 上有三个组件设置了约束。最上面的是专辑的封面，封面下面是列举了相关专辑的列表，最下面是有两个按钮的工具栏，一个用来撤销操作，另一个用来删除你选中的专辑。 StoryBoard 看起来是这个样子的：



- 一个简单的 HTTP 客户端类 （HTTPClient)，里面还没有什么内容，需要你去完善。

> 注意：其实当你创建一个新的 Xcode 的项目的时候，你的代码里就已经有很多设计模式的影子了： MVC、委托、代理、单例 - 真是众里寻他千百度，得来全不费功夫。

在学习第一个设计模式之前，你需要创建两个类，用来存储和展示专辑数据。

创建一个新的类，继承 `NSObject` 名为 `Album` ，记得选择 Swift 作为编程语言然后点击下一步。

打开 `Album.swift` 然后添加如下定义：

```
var title : String!
var artist : String!
var genre : String!
var coverUrl : String!
var year : String!
```

这里创建了五个属性，分别对应专辑的标题、作者、流派、封面地址和出版年份。

接下来我们添加一个初始化方法：

```
init(title: String, artist: String, genre: String, coverUrl: String
  super.init()
  self.title = title
  self.artist = artist
  self.genre = genre
  self.coverUrl = coverUrl
  self.year = year
}
```

这样我们就可以愉快的初始化了。

然后再加上下面这个方法：

```
override var description: String {
    return "title: \(title)" +
           "artist: \(artist)" +
           "genre: \(genre)" +
           "coverUrl: \(coverUrl)" +
           "year: \(year)"
}
```

这是专辑对象的描述方法，详细的打印了 `Album` 的所有属性值，方便我们查看变量各个属性的值。

接下来，再创建一个继承自 `UIView` 的视图类 `AlbumView.swift` 。

在新建的类中添加两个属性：

```
private var coverImage: UIImageView!
private var indicator: UIActivityIndicatorView!
```

`coverImage` 代表了封面的图片， `indicator` 则是在加载过程中显示的等待指示器。

这两个属性都是私有属性，因为除了 `AlbumView` 之外，其他类没有必要知道他俩的存在。在写一些框架或者类库的时候，这种规范十分重要，可以避免一些误操作。

接下来给这个类添加初始化化方法：

```
required init?(coder aDecoder: NSCoder) {
    super.init(coder: aDecoder)!
}

init(frame: CGRect, albumCover: String) {
    super.init(frame: frame)
    backgroundColor = UIColor.blackColor()
    coverImage = UIImageView(frame: CGRectMake(5, 5, frame.size.wic
    addSubview(coverImage)
    indicator = UIActivityIndicatorView()
    indicator.center = center
    indicator.activityIndicatorViewStyle = .WhiteLarge
    indicator.startAnimating()
    addSubview(indicator)
}
```

因为 `UIView` 遵从 `NSCoding` 协议，所以我们需要 `NSCoder` 的初始化方法。不过目前我们没有 `encode` 和 `decode` 的必要，所以就把它放在那里就行，调用父类方法初始化即可。

在真正的初始化方法里，我们设置了一些初始化的默认值。比如设置背景颜色默认为黑色，创建 `ImageView` 并设置了 `margin` 值，添加了一个加载指示器。

最终我们再加上如下方法：

```
func highlightAlbum(didHighlightView didHighlightView: Bool) {
    if didHighlightView == true {
        backgroundColor = UIColor.whiteColor()
    } else {
        backgroundColor = UIColor.blackColor()
    }
}
```

这会切换专辑的背景颜色，如果高亮就是白色，否则就是黑色。

在继续下面的内容之前， `Command + B` 试一下确保没有什么问题，一切正常？那就开始第一个设计模式的学习啦！

完成到这一步的Demo：

- 查看源码
- 下载Zip

设计模式之王 - MVC



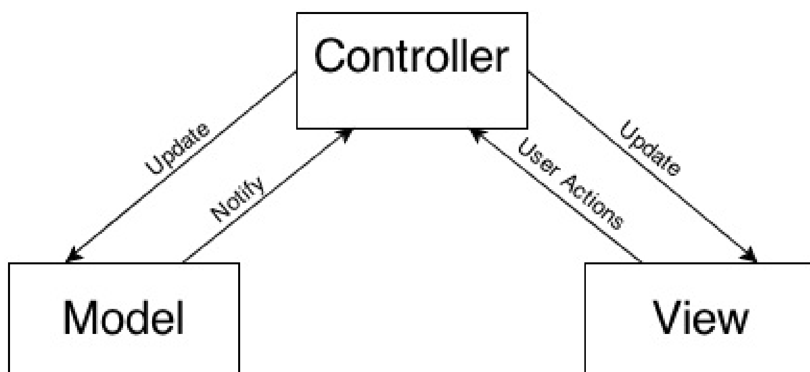
`Model-View-Controller` (缩写 MVC) 是 Cocoa 框架的一部分，并且毋庸置疑是最常用的设计模式之一。它可以帮你把对象根据职责进行划分和归类。

作为划分依据的三个基本职责是：

- 模型层 (Model) ：存储数据并且定义如何操作这些数据。在我们的例子中，就是 `Album` 类。
- 视图层 (View) ：负责模型层的可视化展示，并且负责用户的交互，一般来说都是继承自 `UIView` 这个基类。在我们的项目中就是 `AlbumView` 这个类。
- 控制器 (Controller) ：控制器是整个系统的掌控者，它连接了模型层和数据层，并且把数据在视图层展示出来，监听各种事件，负责数据的各种操作。不妨猜猜在我们的项目中哪个是控制器？啊哈猜对了： `ViewController` 这个类就是。

如果你的项目遵循 MVC 的设计模式，那么各种对象要不是 Model ，要不是 View ，要不就是 Controller。当然在实际的开发中也可以灵活变化，比如结合具体业务使用 MVVM 结构给 `ViewController` 瘦瘦身，也是可以的。

三者之间的关系如下：



模型层通知控制器层任何数据的变化，然后控制器层会刷新视图层中的数据。视图层可以通知控制器层用户的交互事件，然后控制器会处理各种事件以及刷新数据。

你可能会感觉奇怪：为什么要把这三个东西分开来，而不能揉在一个类里呢？那样似乎更简单一点嘛。

之所以这样做，是为了将代码更好的分离和重用。理想状态下，视图层应当和模型层完全分离。如果视图层不依赖任何模型层的具体实现，那么就可以很容易的被其他模型复用，用来展示不同的数据。

举个例子，比如在未来我们需要添加电影或者什么书籍，我们依旧可以使用 `AlbumView` 这个类作为展示。更久远点来说，在以后如果你创建了一个新的项目并且需要用到和专辑相关的内容，你可以直接复用 `Album` 类因为它并不依赖于任何视图模块。这就是 MVC 的强大之处，三大元素，各司其职，减少依赖。

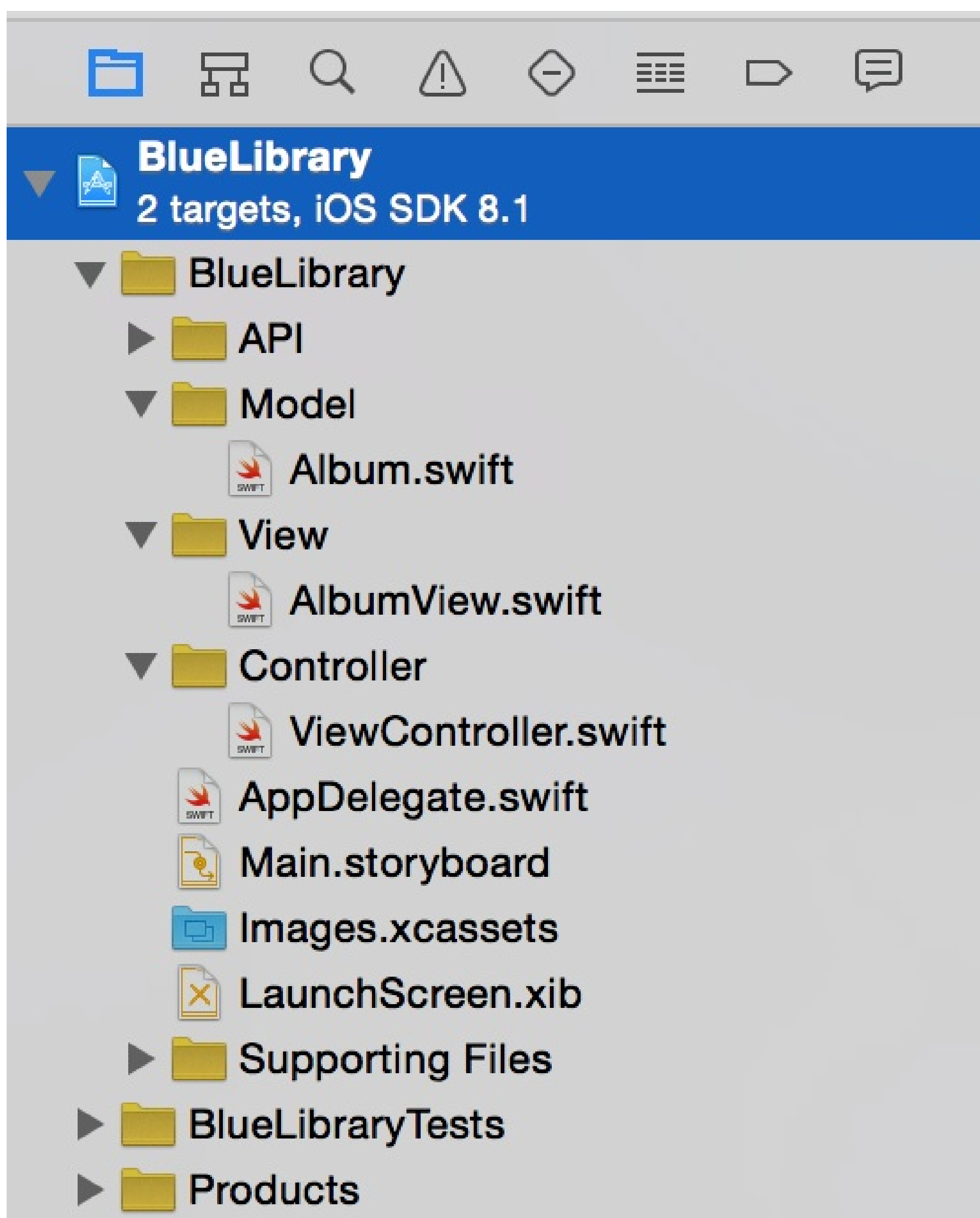
如何使用 MVC 模式

首先，你需要确定你的项目中的每个类都是三大基本类型中的一种：控制器、模型、视图。不要在一个类里糅合多个角色。目前我们创建了 `Album` 类和 `AlbumView` 类是符合要求的，做得很好。

然后，为了确保你遵循这种模式，你最好创建三个项目分组来存放代码，分别是 Model、View、Controller，保持每个类型的文件分别独立。

接下来把 `Album.swift` 拖到 `Model` 分组，把 `AlbumView.swift` 拖到 `View` 分组，然后把 `ViewController.swift` 拖到 `Controller` 分组中。

现在你的项目应该是这个样子：



现在你的项目已经有点样子了，不再是各个文件颠沛流离居无定所了。显然你还会有其他分组和类，但是应用的核心就在这三个类里。

现在你的内容已经组织好了，接下来要做的就是获取专辑的数据。你将会创建一个 API 类来管理数据 - 这里我们会用到下一个设计模式：单例模式。

单例模式 - Singleton

单例模式确保每个指定的类只存在一个实例对象，并且可以全局访问那个实例。一般情况下会使用延时加载的策略，只在第一次需要使用的时候初始化。

注意：在 iOS 中单例模式很常见， `NSUserDefaults.standardUserDefaults()` 、 `UIApplication.sharedApplication()` 、 `UIScreen.mainScreen()` 、 `NSFileManager.defaultManager()` 这些都是单例模式。

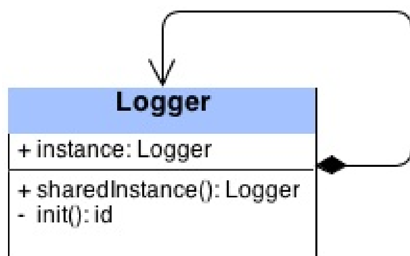
你可能会疑惑了：如果多于一个实例又会怎么样呢？代码和内存还没精贵到这个地步吧？

某些场景下，保持实例对象仅有一份是很有意义的。举个例子，你的应用实例(`UIApplication`)，应该只有一个吧，显然是指你的当前应用。还有一个例子：设备的屏幕(`UIScreen`)实例也是这样，所以对于这些类的情况，你只想要一个实例对象。

单例模式的应用还有另一种情况：你需要一个全局类来处理配置文件。我们很容易通过单例模式实现线程安全的实例访问，而如果有多个类可以同时访问配置文件，那可就复杂多了。

如何使用单例模式

可以看下这个图：



这是一个日志类，有一个属性 (是一个单例对象) 和两个方法 (`sharedInstance()` 和 `init()`)。

第一次调用 `sharedInstance()` 的时候， `instance` 属性还没有初始化。所以我们要创建一个新实例并且返回。

下一次你再调用 `sharedInstance()` 的时候， `instance` 已经初始化完成，直接返回即可。这个逻辑确保了这个类只存在一个实例对象。

接下来我们继续完善单例模式，通过这个类来管理专辑数据。

注意到在我们前面的截图里，分组中有个 `API` 分组，这里可以放那些提供后台服务的类。在这个分组中创建一个新的文件 `LibraryAPI.swift` ，继承自 `NSObject` 类。

在 `LibraryAPI` 里添加下面这段代码：

```
//1
class var sharedInstance: LibraryAPI {
    //2
    struct Singleton {
        //3
        static let instance = LibraryAPI()
    }
    //4
    return Singleton.instance
}
```

在这几行代码里，做了如下工作：

创建一个计算类型的类变量，这个类变量，就像是 objc 中的静态方法一样，可以直接通过类访问而不用实例对象。具体可参见苹果官方文档的 属性 这一章。

在类变量里嵌套一个 `Singleton` 结构体。

`Singleton` 封装了一个静态的常量，通过 `static` 定义意味着这个属性只存在一个，注意 Swift 中 `static` 的变量是延时加载的，意味着 `Instance` 直到需要的时候才会被创建。

同时再注意一下，因为它是一个常量，所以一旦创建之后不会再创建第二次。这些就是单例模式的核心所在：一旦初始化完成，当前类存在一个实例对象，初始化方法就不会再被调用。

返回计算后的属性值。

注意：更多的单例模式实例可以看看 Github 上的这个示例，列举了单例模式的若干种实现方式。

你现在可以将这个单例作为专辑管理类的入口，接下来我们继续创建一个处理专辑数据持久化的类。

新建 `PersistencyManager.swift` 并添加如下代码：

```
private var albums = [Album]()
```

在这里我们定义了一个私有属性，用来存储专辑数据。这是一个可变数组，所以你可以很容易的增加或者删除数据。

然后加上一些初始化的数据：

```
override init() {
  //Dummy list of albums
  let album1 = Album(title: "Best of Bowie",
      artist: "David Bowie",
      genre: "Pop",
      coverUrl: "http://img3.douban.com/mpic/s1497881.jpg",
      year: "1992")

  let album2 = Album(title: "It's My Life",
      artist: "No Doubt",
      genre: "Pop",
      coverUrl: "http://img3.doubanio.com/mpic/s3880529.jpg",
      year: "2003")

  let album3 = Album(title: "Nothing Like The Sun",
      artist: "Sting",
      genre: "Pop",
      coverUrl: "http://img3.doubanio.com/mpic/s3708339.jpg",
      year: "1999")

  let album4 = Album(title: "Staring at the Sun",
      artist: "U2",
      genre: "Pop",
      coverUrl: "http://img3.douban.com/mpic/s1882422.jpg",
      year: "2000")

  let album5 = Album(title: "American Pie",
      artist: "Madonna",
      genre: "Pop",
      coverUrl: "http://img3.douban.com/mpic/s3105351.jpg",
      year: "2000")

  albums = [album1, album2, album3, album4, album5]
}
```

在这个初始化方法里，我们初始化了五张专辑。如果上面的专辑没有你喜欢的，你可以随意替换成你的菜:]

然后添加如下方法：

```
func getAlbums() -> [Album] {
  return albums
}

func addAlbum(album: Album, index: Int) {
  if (albums.count >= index) {
    albums.insert(album, atIndex: index)
  } else {
    albums.append(album)
  }
}

func deleteAlbumAtIndex(index: Int) {
  albums.removeAtIndex(index)
}
```

这些方法可以让你自由的访问、添加、删除专辑数据。

这时你可以运行一下你的项目，确保编译通过以便进行下一步操作。

此时你或许会感到好奇：`PersistencyManager` 好像不是单例啊？是的，它确实不是单例。不过没关系，在接下来的外观模式章节，你会看到 `LibraryAPI` 和 `PersistencyManagerx` 之间的联系。

完成到这一步的Demo：

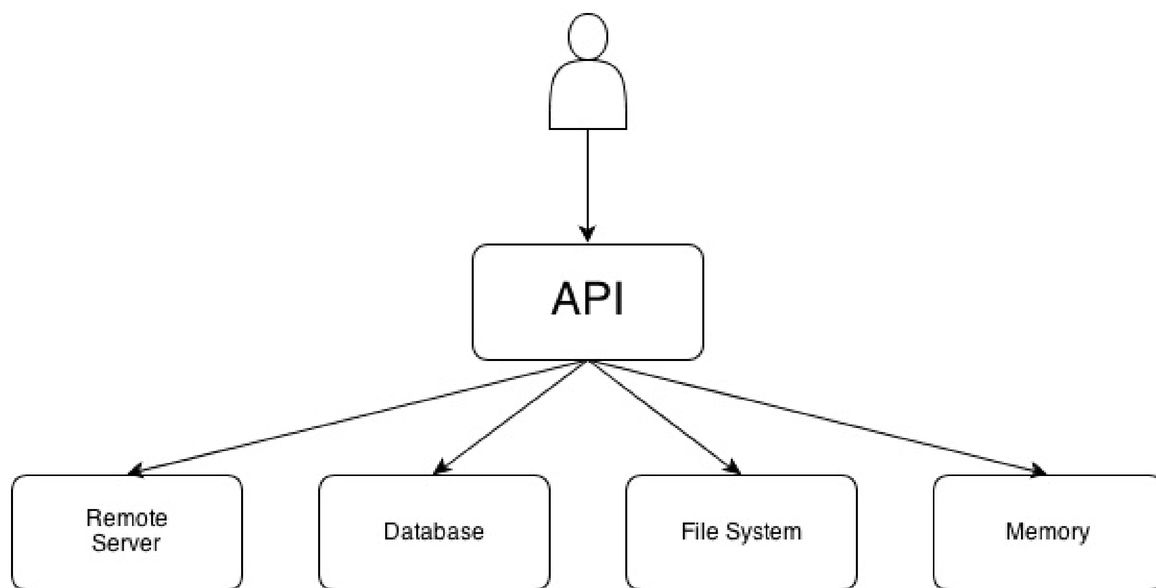
- 查看源码
- 下载Zip

外观模式 - Facade



外观模式在复杂的业务系统上提供了简单的接口。如果直接把业务的所有接口直接暴露给使用者，使用者需要单独面对这一大堆复杂的接口，学习成本很高，而且存在误用的隐患。如果使用外观模式，我们只要暴露必要的 API 就可以了。

下图演示了外观模式的基本概念：



API 的使用者完全不知道这内部的业务逻辑有多么复杂。当我们有大量的类并且它们使用起来很复杂而且也很难理解的时候，外观模式是一个十分理想的选择。

外观模式把使用和背后的实现逻辑成功解耦，同时也降低了外部代码对内部工作的依赖程度。如果底层的类发生了改变，外观的接口并不需要做修改。

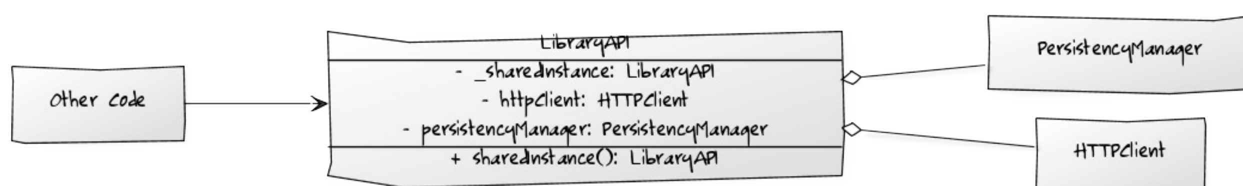
举个例子，如果有一天你想换掉所有的后台服务，你只需要修改 API 内部的代码，外部调用 API 的代码并不会有改动。

如何使用外观模式

现在我们用 `PersistencyManager` 来管理专辑数据，用 `HTTPClient` 来处理网络请求，项目中的其他类不应该知道这个逻辑。他们只需要知道 `LibraryAPI` 这个“外观”就可以了。

为了实现外观模式，应该只让 `LibraryAPI` 持有 `PersistencyManager` 和 `HTTPClient` 的实例，然后 `LibraryAPI` 暴露一个简单的接口给其他类来访问，这样外部的访问类不需要知道内部的业务具体是怎样的，也不用知道你是通过 `PersistencyManager` 还是 `HTTPClient` 获取到数据的。

大致的设计是这样的：



`LibraryAPI` 会暴露给其他代码访问，但是 `PersistencyManager` 和 `HTTPClient` 则是不对外开放的。

打开 `LibraryAPI.swift` 然后添加如下代码：

```
private let persistencyManager: PersistencyManager
private let httpClient: HTTPClient
private let isOnline: Bool
```

除了两个实例变量之外，还有个 `Bool` 值： `isOnline` ，这个是用来标识当前是否为联网状态的，如果是联网状态就会去网络获取数据。

我们需要在 `init` 里面初始化这些变量：

```
override init() {
  persistencyManager = PersistencyManager()
  httpClient = HTTPClient()
  isOnline = false

  super.init()
}
```

`HTTPClient` 并不会直接和真实的服务器交互，只是用来演示外观模式的使用。所以 `inOnline` 这个值我们一直设置为 `false` 。

接下来在 `LibraryAPI.swift` 里添加如下代码：

```
func getAlbums() -> [Album] {
  return persistencyManager.getAlbums()
}

func addAlbum(album: Album, index: Int) {
  persistencyManager.addAlbum(album, index: index)
  if isOnline {
    httpClient.postRequest("/api/addAlbum", body: album.descriptio
  }
}

func deleteAlbum(index: Int) {
  persistencyManager.deleteAlbumAtIndex(index)
  if isOnline {
    httpClient.postRequest("/api/deleteAlbum", body: "\(index)")
  }
}
```

看一下 `addAlbum(_:index:)` 这个方法，先更新本地缓存，然后如果是联网状态还需要向服务器发送网络请求。这便是外观模式的强大之处：如果外部文件想要添加一个新的专辑，它不会也不用去了解内部的实现逻辑是怎么样的。

注意：当你设计外观的时候，请务必牢记：使用者随时可能直接访问你的隐藏类。永远不要假设使用者会遵循你当初的设计做事。

运行一下你的应用，你可以看到两个空的页面和一个工具栏：最上面的视图用来展示专辑封面，下面的视图展示数据列表。


Carrier
12:33 AM
Pop Music

你需要在屏幕上展示专辑数据，这是就该用下一种设计模式了：装饰者模式。

完成到这一步的Demo：

- 查看源码
- 下载Zip

装饰者模式 - Decorator

装饰者模式可以动态的给指定的类添加一些行为和职责，而不用对原代码进行任何修改。当你需要使用子类的时候，不妨考虑一下装饰者模式，可以在原始类上面封装一层。

在 Swift 里，有两种方式实现装饰者模式：扩展 (Extension) 和委托 (Delegation)。

扩展

扩展是一种十分强大的机制，可以让你在不用继承的情况下，给已存在的类、结构体或者枚举类添加一些新的功能。最重要的一点是，你可以在你没有访问权限的情况下扩展已有类。这意味着你甚至可以扩展 Cocoa 的类，比如 `UIView` 或者 `UIImage` 。

举个例子，在编译时新加的方法可以像扩展类的正常方法一样执行。这和装饰器模式有点不同，因为扩展不会持有扩展类的对象。

如何使用扩展

想象一下这个场景，我们需要在下面这个列表里展示数据：

Artist	David Bowie
Album	Best of Bowie
Genre	Pop
Year	1992

专辑标题从哪里来？ `Album` 本身是个 Model 对象，所以它不应该负责如何展示数据。你需要一些额外的代码添加展示数据的逻辑，但是为了保持 `Model` 的干净，我们不应该直接修改代码，因为这样不符合单一职责原则。 `Model` 层最好就是负责纯粹的数据结构，如果有数据的操作可以放到扩展中完成。

接下来我们会创建一个扩展，扩展现有的 `Album` 类，在扩展里定义了新的方法，返回更适合 `UITableView` 展示用的数据结构。

数据的结构大概是这样：

titles →	"Artist"	"Album"	"Genre"	"Year"
values →	album.artist	album.title	album.genre	album.year

新建一个 Swift 文件： `AlbumExtensions` ，在里面添加如下扩展：

```
extension Album {
  func ae_tableRepresentation() -> (titles:[String], values:[String
    return (["Artist", "Album", "Genre", "Year"], [artist, title, g
  }
}
```

在方法的前面有个 `ae_` 前缀，是 `AlbumExtension` 的缩写，这样有利于和类的原有方法进行区分，避免使用的时候产生冲突。现在很多还在维护中的第三方库都已经改成了这个风格。

注意：类是可以重写父类方法的，但是在扩展里不可以。扩展里的方法和属性不能和原始类里的方法和属性冲突。

思考一下这个设计模式的强大之处：

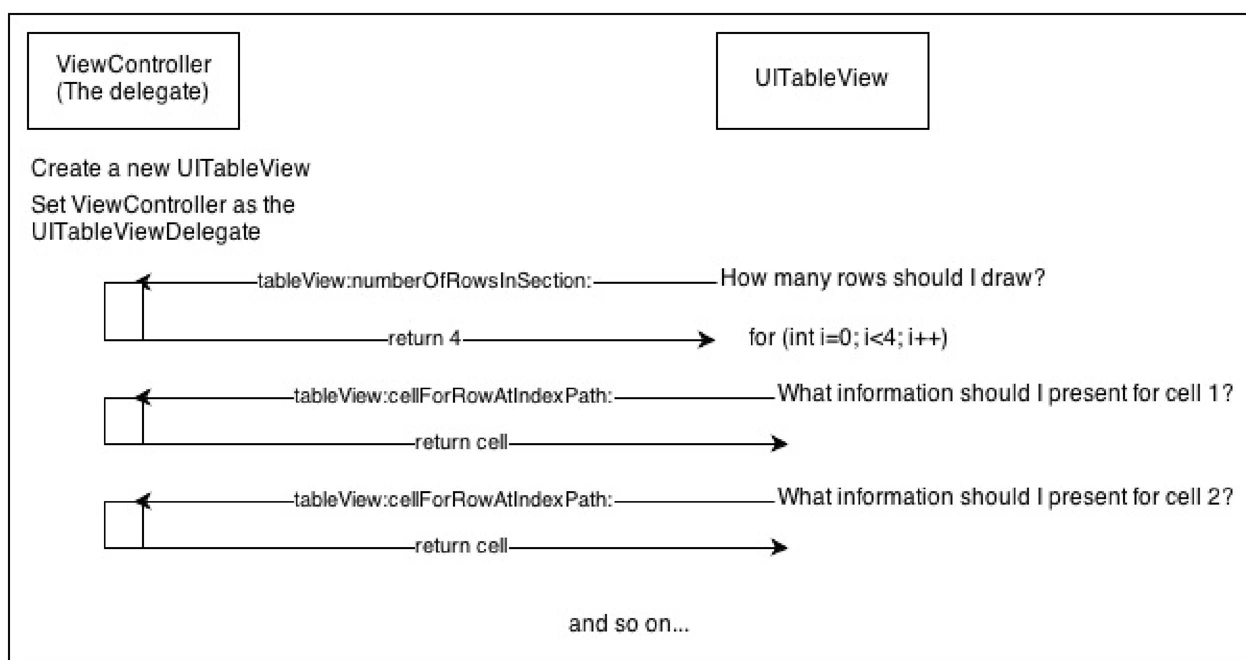
- 我们可以直接在扩展里使用 `Album` 里的属性。
- 我们给 `Album` 类添加了内容但是并没有继承它，事实上，使用继承来扩展业务也可以实现一样的功能。
- 这个简单的扩展让我们可以更好地把 `Album` 的数据展示在 `UITableView` 里，而且不用修改源码。

委托

装饰者模式的另一种实现方案是委托。在这种机制下，一个对象可以和另一个对象相关联。比如你在用 `UITableView` ，你必须实现 `tableView(_:numberOfRowsInSection:)` 这个委托方法。

你不应该指望 `UITableView` 知道你有多少数据，这是个应用层该解决的问题。所以，数据相关的计算应该通过 `UITableView` 的委托来解决。这样可以让 `UITableView` 和数据层分别独立。视图层就负责显示数据，你递过来什么我就显示什么。

下面这张图很好的解释了 `UITableView` 的工作过程：



`UITableView` 的工作仅仅是展示数据，但是最终它需要知道自己要展示那些数据，这时就可以向它的委托询问。在 objc 的委托模式里，一个类可以通过协议来声明可选或者必须的方法。

看起来似乎继承然后重写必须的方法来的更简单一点。但是考虑一下这个问题：继承的结果必定是一个独立的类，如果你想让某个对象成为多个对象的委托，那么子类这招就行不通了。

注意：委托模式十分重要，苹果在 UIKit 中大量使用了该模式，基本上随处可见。

如何使用委托模式

打开 `ViewController.swift` 文件，添加如下私有变量：

```
private var allAlbums = [Album]()
private var currentAlbumData : (titles:[String], values:[String])?
private var currentAlbumIndex = 0
```

在 `viewDidLoad` 里面加入如下内容：

```
override func viewDidLoad() {
    super.viewDidLoad()
    //1
    self.navigationController?.navigationBar.translucent = false
    currentAlbumIndex = 0

    //2
    allAlbums = LibraryAPI.sharedInstance.getAlbums()

    // 3
    // the uitableview that presents the album data
    dataTable.delegate = self
    dataTable.dataSource = self
    dataTable.backgroundView = nil
    view.addSubview(dataTable!)
}
```

对上面三个部分进行拆解：

1. 关闭导航栏的透明效果
2. 通过 API 获取所有的专辑数据，记住，我们使用外观模式之后，应该从 `LibraryAPI` 获取数据，而不是 `PersistencyManager` 。
3. 你可以在这里设置你的 `UITablweView` ，在这里声明了 `UITableView` 的 `delegate` 是当前的 `ViewController` 。事实上你用了 XIB 或者 StoryBoard，可以直接在可视化的页面里拖拽完成。

接下来添加一个新的方法用来更方便的获取数据：

```
func showDataForAlbum(albumIndex: Int) {
    // defensive code: make sure the requested index is lower than
    if (albumIndex < allAlbums.count && albumIndex > -1) {
        //fetch the album
        let album = allAlbums[albumIndex]
        // save the albums data to present it later in the tablevie
        currentAlbumData = album.ae_tableRepresentation()
    } else {
        currentAlbumData = nil
    }
    // we have the data we need, let's refresh our tableview
    dataTable!.reloadData()
}
```

`showDataForAlbum()` 这个方法获取最新的专辑数据，当你想要展示新数据的时候，你需要调用 `reloadData()` 这个方法，这样 `UITableView` 就会向委托请求数据，比如有多少个 `section` 有多少个 `row` 之类的。

在 `viewDidLoad` 里面调用上面的方法：

```
self.showDataForAlbum(currentAlbumIndex)
```

这样应用一启动就会去加载当前的专辑数据。因为 `currentAlbumIndex` 的默认值是 0，所以一开始会默认显示第一章专辑的信息。

接下来我们该去完善 `DataSource` 的协议方法了。你可以直接把委托方法写在类里面，当然如果你想让你的代码看起来更整洁一点，则可以放在扩展里。

在文件底部添加如下方法，注意一定要放在类定义的大括号外面，因为这两个家伙不是类定义的一部分，它们是扩展：

```
extension ViewController: UITableViewDataSource {
}

extension ViewController: UITableViewDelegate {
}
```

上面就是实现委托的方法 - 你可以把协议想象成是与委托之间的约定，只要你实现了约定的方法，就算是实现了委托。在我们的代码中， `ViewController` 需要遵守 `UITableViewDataSource` 和 `UITableViewDelegate` 的协议。这样 `UITableView` 才能确保必要的委托方法都已经实现了。

在 `UITableViewDataSource` 对应的那个扩展里加上如下方法：

```
func tableView(tableView: UITableView, numberOfRowsInSection sectio
  if let albumData = currentAlbumData {
    return albumData.titles.count
  } else {
    return 0
  }
}

func tableView(tableView: UITableView, cellForRowAtIndexPath indexP
  let cell:UITableViewCell = tableView.dequeueReusableCellWithIdent
  if let albumData = currentAlbumData {
    cell.textLabel?.text = albumData.titles[indexPath.row]
      if let detailTextLabel = cell.detailTextLabel {
        detailTextLabel.text = albumData.values[indexPath.row]
      }
  }
  return cell
}
```

`tableView(_:numberOfRowsInSection:)` 返回需要展示的行数，和存储的数据中的 title 的数目相同。

`tableView(_:cellForRowAtIndexPath:)` 创建并且返回了一个单元格，上面有标题和对应的值。

注意：你可以把这些方法直接加在类声明里面，也可以放在扩展里，编译器不会去管数据源到底在哪里，只要能找到对应的方法就可以了。而我们之所以这样做，是为了方便其他人阅读。

此时再构建项目，你可以看到如下内容：

Carrier 12:39 AM

Pop Music

Artist	David Bowie
Album	Best of Bowie
Genre	Pop
Year	1992

是的，显示成功啦！

我们的原计划是在上面的空白处放一个可以横滑浏览专辑的视图。其实仔细想想，这个控件是可以应用在其他地方的，我们不妨把它做成一个可复用的视图。

为了让这个视图可以复用，显示内容的工作都只能交给另一个对象来完成：它的委托。这个横滑页面应该声明一些方法让它的委托去实现，就像是 `UITableView` 的 `UITableViewDelegate` 一样。我们将会在下一个设计模式中实现这个功能。

完成到这一步的Demo：

- 查看源码
- 下载Zip

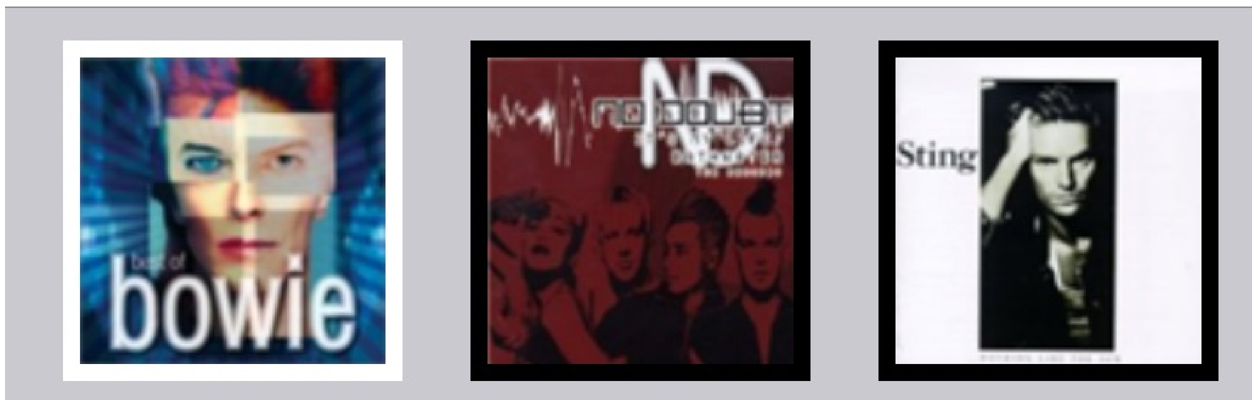
适配器模式 - Adapter

适配器把自己封装起来然后暴露统一的接口给其他类，这样即使其他类的接口各不相同，也能相安无事，一起工作。

如果你熟悉适配器模式，那么你会发现苹果在实现适配器模式的方式稍有不同：苹果通过委托实现了适配器模式。委托相信大家都不陌生。举个例子，如果一个类遵循了 `NSCoying` 的协议，那么它一定要实现 `copy` 方法。

如何使用适配器模式

横滑的滚动栏理论上应该是这个样子的：



新建一个 Swift 文件： `HorizontalScroller.swift` ，作为我们的横滑滚动控件， `HorizontalScroller` 继承自 `UIView` 。

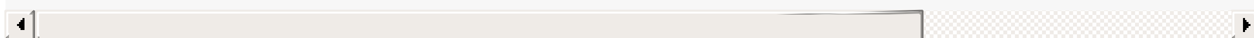
打开 `HorizontalScroller.swift` 文件并添加如下代码：

```
@objc protocol HorizontalScrollerDelegate {
}
```

这行代码定义了一个新的协议： `HorizontalScrollerDelegate` 。我们在前面加上了 `@objc` 的标记，这样我们就可以像在 objc 里一样使用 `@optional` 的委托方法了。

接下来我们在大括号里定义所有的委托方法，包括必须的和可选的：

```
// 在横滑视图中有多少页面需要展示
func numberOfViewsForHorizontalScroller(scroller: HorizontalScrolle
// 展示在第 index 位置显示的 UIView
func horizontalScrollerViewAtIndex(scroller: HorizontalScroller, in
// 通知委托第 index 个视图被点击了
func horizontalScrollerClickedViewAtIndex(scroller: HorizontalScrol
// 可选方法，返回初始化时显示的图片下标，默认是0
optional func initialViewIndex(scroller: HorizontalScroller) -> Int
```



其中，没有 `option` 标记的方法是必须实现的，一般来说包括那些用来显示的必须数据，比如如何展示数据，有多少数据需要展示，点击事件如何处理等等，不可或缺；有 `option` 标记的方法为可选实现的，相当于是一些辅助设置和功能，就算没有实现也有默认值进行处理。

在 `HorizontalScroller` 类里添加一个新的委托对象：

```
weak var delegate: HorizontalScrollerDelegate?
```

为了避免循环引用的问题，委托是 `weak` 类型。如果委托是 `strong` 类型的，当前对象持有了委托的强引用，委托又持有了当前对象的强引用，这样谁都无法释放就会导致内存泄露。

委托是可选类型，所以很有可能当前类的使用者并没有指定委托。但是如果指定了委托，那么它一定会遵循 `HorizontalScrollerDelegate` 里约定的内容。

再添加一些新的属性：

```
// 1
private let VIEW_PADDING = 10
private let VIEW_DIMENSIONS = 100
private let VIEWS_OFFSET = 100

// 2
private var scroller : UIScrollView!
// 3
var viewArray = [UIView]()
```

上面标注的三点分别做了这些事情：

- 定义一个常量，用来方便的改变布局。现在默认的是显示的内容长宽为100，间隔为10。
- 创建一个 `UIScrollView` 作为容器。
- 创建一个数组用来存放需要展示的数据

接下来实现初始化方法：

```
override init(frame: CGRect) {
    super.init(frame: frame)
    initializeScrollView()
}

required init(coder aDecoder: NSCoder) {
    super.init(coder: aDecoder)
    initializeScrollView()
}

func initializeScrollView() {
    //1
    scroller = UIScrollView()
    scroller.delegate = self
    addSubview(scroller)

    //2
    scroller.translatesAutoresizingMaskIntoConstraints = false

    //3
    self.addConstraint(NSLayoutConstraint(item: scroller, attribute
    self.addConstraint(NSLayoutConstraint(item: scroller, attribute
    self.addConstraint(NSLayoutConstraint(item: scroller, attribute
    self.addConstraint(NSLayoutConstraint(item: scroller, attribute

    //4
    let tapRecognizer = UITapGestureRecognizer(target: self, action
    scroller.addGestureRecognizer(tapRecognizer)
}
```

上面的代码做了如下工作：

- 创建一个 `UIScrollView` 对象并且把它加到父视图中。
- 关闭 `autoresizing masks` ，从而可以使用 `AutoLayout` 进行布局。
- 给 `scrollview` 添加约束。我们希望 `scrollview` 能填满 `HorizontalScroller` 。
- 创建一个点击事件，检测是否点击到了专辑封面，如果确实点击到了专辑封面，我们需要通知 `HorizontalScroller` 的委托。

添加委托方法：

```
func scrollerTapped(gesture: UITapGestureRecognizer) {
  let location = gesture.locationInView(gesture.view)
  if let delegate = self.delegate {
    for index in 0..<delegate.numberOfViewsForHorizontalScroller(se
      let view = scroller.subviews[index] as UIView
      if CGRectContainsPoint(view.frame, location) {
        delegate.horizontalScrollerClickedViewAtIndex(self, index:
        scroller.setContentOffset(CGPointMake(view.frame.origin.x -
        break
      }
    }
  }
}
```

我们把 `gesture` 作为一个参数传了进来，这样就可以获取点击的具体坐标了。

接下来我们调用了 `numberOfViewsForHorizontalScroller` 方法， `HorizontalScroller` 不知道自己的 `delegate` 具体是谁，但是知道它一定实现了 `HorizontalScrollerDelegate` 协议，所以可以放心的调用。

对于 `scroll view` 中的 `view` ，通过 `CGRectContainsPoint` 进行点击检测，从而获知是哪一个 `view` 被点击了。当找到了点击的 `view` 的时候，则会调用委托方法里的 `horizontalScrollerClickedViewAtIndex` 方法通知委托。在跳出 `for` 循环之前，先把点击到的 `view` 居中。

接下来我们再加个方法获取数组里的 `view` ：

```
func viewAtIndex(index :Int) -> UIView {
  return viewArray[index]
}
```

这个方法很简单，只是用来更方便获取数组里的 `view` 而已。在后面实现高亮选中专辑的时候会用到这个方法。

添加如下代码用来重新加载 `scroller` ：

```
func reload() {
  // 1 - Check if there is a delegate, if not there is nothing to l
  if let delegate = self.delegate {
    //2 - Will keep adding new album views on reload, need to reset
    viewArray = []
    let views: NSArray = scroller.subviews

    // 3 - remove all subviews
    views.enumerateObjectsUsingBlock {
    (object: AnyObject!, idx: Int, stop: UnsafeMutablePointer<ObjCB
      object.removeFromSuperview()
    }
    // 4 - xValue is the starting point of the views inside the scr
    var xValue = VIEWS_OFFSET
    for index in 0..<delegate.numberOfViewsForHorizontalScroller(se
      // 5 - add a view at the right position
      xValue += VIEW_PADDING
      let view = delegate.horizontalScrollerViewAtIndex(self, index
      view.frame = CGRectMake(CGFloat(xValue), CGFloat(VIEW_PADDING
      scroller.addSubview(view)
      xValue += VIEW_DIMENSIONS + VIEW_PADDING
      // 6 - Store the view so we can reference it later
     viewArray.append(view)
    }
    // 7
    scroller.contentSize = CGSizeMake(CGFloat(xValue + VIEWS_OFFSET

    // 8 - If an initial view is defined, center the scroller on it
    if let initialView = delegate.initialViewIndex?(self) {
        let xFinal = CGFloat(initialView) * CGFloat(VIEW_DIMENSIONS
        scroller.setContentOffset(CGPointMake(xFinal, 0), animated:
    }
  }
}
```

这个 `reload` 方法有点像是 `UITableView` 里面的 `reloadData` 方法，它会重新加载所有数据。

一段一段的看下上面的代码：

- 在调用 `reload` 之前，先检查一下是否有委托。
- 既然要清除专辑封面，那么也需要重新设置 `viewArray` ，要不然以前的数据会累加进来。
- 移除先前加入到 `scrollview` 的子视图。
- 所有的 `view` 都有一个偏移量，目前默认是100，我们可以修改 `VIEW_OFFSET` 这个常量轻松的修改它。
- `HorizontalScroller` 通过委托获取对应位置的 `view` 并且把它们放在对应的位置上。
- 把 `view` 存进 `viewArray` 以便后面的操作。
- 当所有 `view` 都安放好了，再设置一下 `content size` 这样才可以进行滑动。
- `HorizontalScroller` 检查一下委托是否实现了 `initialViewIndex()` 这个可选方法，这种检查十分必要，因为这个委托方法是可选的，如果委托没有实现这个方法则用0作为默认值。最终设置 `scroll view` 将初始的 `view` 放置到居中的位置。

当数据发生改变的时候，我们需要调用 `reload` 方法。当 `HorizontalScroller` 被加到其他页面的时候也需要调用这个方法，我们在 `HorizontalScroller.swift` 里面加入如下代码：

```
override func didMoveToSuperview() {
    reload()
}
```

在当前 `view` 添加到其他 `view` 里的时候就会自动调用 `didMoveToSuperview` 方法，这样可以在正确的时间重新加载数据。

`HorizontalScroller` 的最后一部分是用来确保当前浏览的内容时刻位于正中心的位置，为了实现这个功能我们需要在用户滑动结束的时候做一些额外的计算和修正。

添加下面这个方法：

```
func centerCurrentView() {
    var xFinal = scroller.contentOffset.x + CGFloat((VIEWS_OFFSET/2
    let viewIndex = xFinal / CGFloat((VIEW_DIMENSIONS + (2*VIEW_PAD
    xFinal = viewIndex * CGFloat(VIEW_DIMENSIONS + (2*VIEW_PADDING)
    scroller.setContentOffset(CGPointMake(xFinal, 0), animated: tru
    if let delegate = self.delegate {
        delegate.horizontalScrollerClickedViewAtIndex(self, index:
    }
}
```

上面的代码计算了当前视图里中心位置距离多少，然后算出正确的居中坐标并滑动到那个位置。最后一行是通知委托所选视图已经发生了改变。

为了检测到用户滑动的结束时间，我们还需要实现 `UIScrollViewDelegate` 的方法。在文件结尾加上下面这个扩展：

```
extension HorizontalScroller: UIScrollViewDelegate {
    func scrollViewDidEndDragging(scrollView: UIScrollView, willDec
        if !decelerate {
            centerCurrentView()
        }
    }

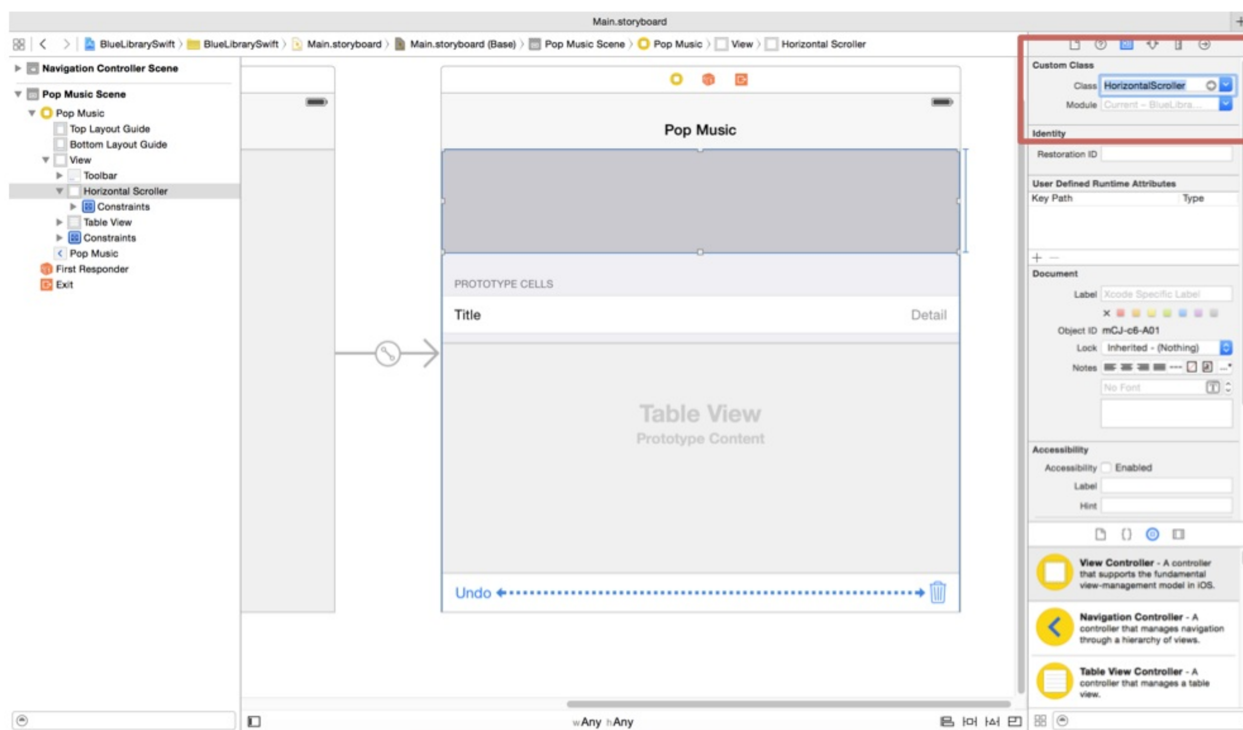
    func scrollViewDidEndDecelerating(scrollView: UIScrollView) {
        centerCurrentView()
    }
}
```

当用户停止滑动的时候， `scrollViewDidEndDragging(_:willDecelerate:)` 这个方法会通知委托。如果滑动还没有停止， `decelerate` 的值为 `true` 。当滑动完全结束的时候，则会调用 `scrollViewDidEndDecelerating` 这个方法。在这两种情况下，你都应该把当前的视图居中，因为用户的操作可能会改变当前视图。

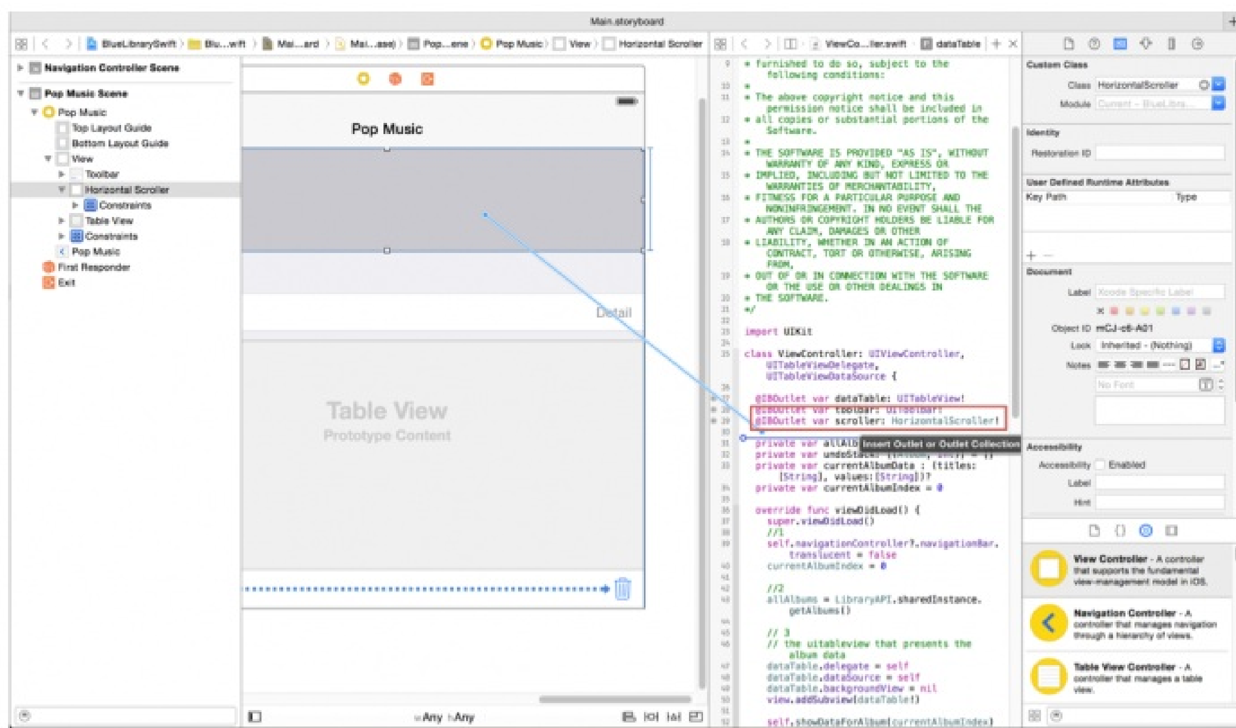
你的 `HorizontalScroller` 已经可以使用了！回头看看前面写的代码，你会看到我们并没有涉及什么 `Album` 或者 `AlbumView` 的代码。这是极好的，因为这样意味着这个 `scroller` 是完全独立的，可以复用。

运行一下你的项目，确保编译通过。

这样，我们的 `HorizontalScroller` 就完成了，接下来我们就要把它应用到我们的项目里了。首先，打开 `Main.Sstoryboard` 文件，点击上面的灰色矩形，设置 `Class` 为 `HorizontalScroller` ：



接下来，在 `assistant editor` 模式下向 `ViewController.swift` 拖拽生成 outlet，命名为 `scroller` ：



接下来打开 `ViewController.swift` 文件，是时候实现 `HorizontalScrollerDelegate` 委托里的方法啦！

添加如下扩展：

```
extension ViewController: HorizontalScrollerDelegate {
    func horizontalScrollerClickedViewAtIndex(scroller: HorizontalS
        //1
        let previousAlbumView = scroller.viewAtIndex(currentAlbumIn
        previousAlbumView.highlightAlbum(didHighlightView: false)
        //2
        currentAlbumIndex = index
        //3
        let albumView = scroller.viewAtIndex(index) as AlbumView
        albumView.highlightAlbum(didHighlightView: true)
        //4
        showDataForAlbum(index)
    }
}
```

让我们一行一行的看下这个委托的实现：

- 获取上一个选中的相册，然后取消高亮

- 存储当前点击的相册封面
- 获取当前选中的相册，设置为高亮
- 在 `table view` 里面展示新数据

接下来在扩展里添加如下方法：

```
func numberOfViewsForHorizontalScroller(scroller: HorizontalScrolle
    return allAlbums.count
}
```

这个委托方法返回 `scroll vew` 里面的视图数量，因为是用来展示所有的专辑的封面，所以数目也就是专辑数目。

然后添加如下代码：

```
func horizontalScrollerViewAtIndex(scroller: HorizontalScroller, in
    let album = allAlbums[index]
    let albumView = AlbumView(frame: CGRectMake(0, 0, 100, 100), al
    if currentAlbumIndex == index {
        albumView.highlightAlbum(didHighlightView: true)
    } else {
        albumView.highlightAlbum(didHighlightView: false)
    }
    return albumView
}
```

我们创建了一个新的 `AlbumView` ，然后检查一下是不是当前选中的专辑，如果是则设为高亮，最后返回结果。

是的就是这么简单！三个方法，完成了一个横向滚动的浏览视图。

我们还需要创建这个滚动视图并把它加到主视图里，但是在这之前，先添加如下方法：

```
func reloadScroller() {
    allAlbums = LibraryAPI.sharedInstance.getAlbums()
    if currentAlbumIndex < 0 {
        currentAlbumIndex = 0
    } else if currentAlbumIndex >= allAlbums.count {
        currentAlbumIndex = allAlbums.count - 1
    }
    scroller.reload()
    showDataForAlbum(currentAlbumIndex)
}
```

这个方法通过 `LibraryAPI` 加载专辑数据，然后根据 `currentAlbumIndex` 的值设置当前视图。在设置之前先进行了校正，如果小于0则设置第一个专辑为展示的视图，如果超出了范围则设置最后一个专辑为展示的视图。

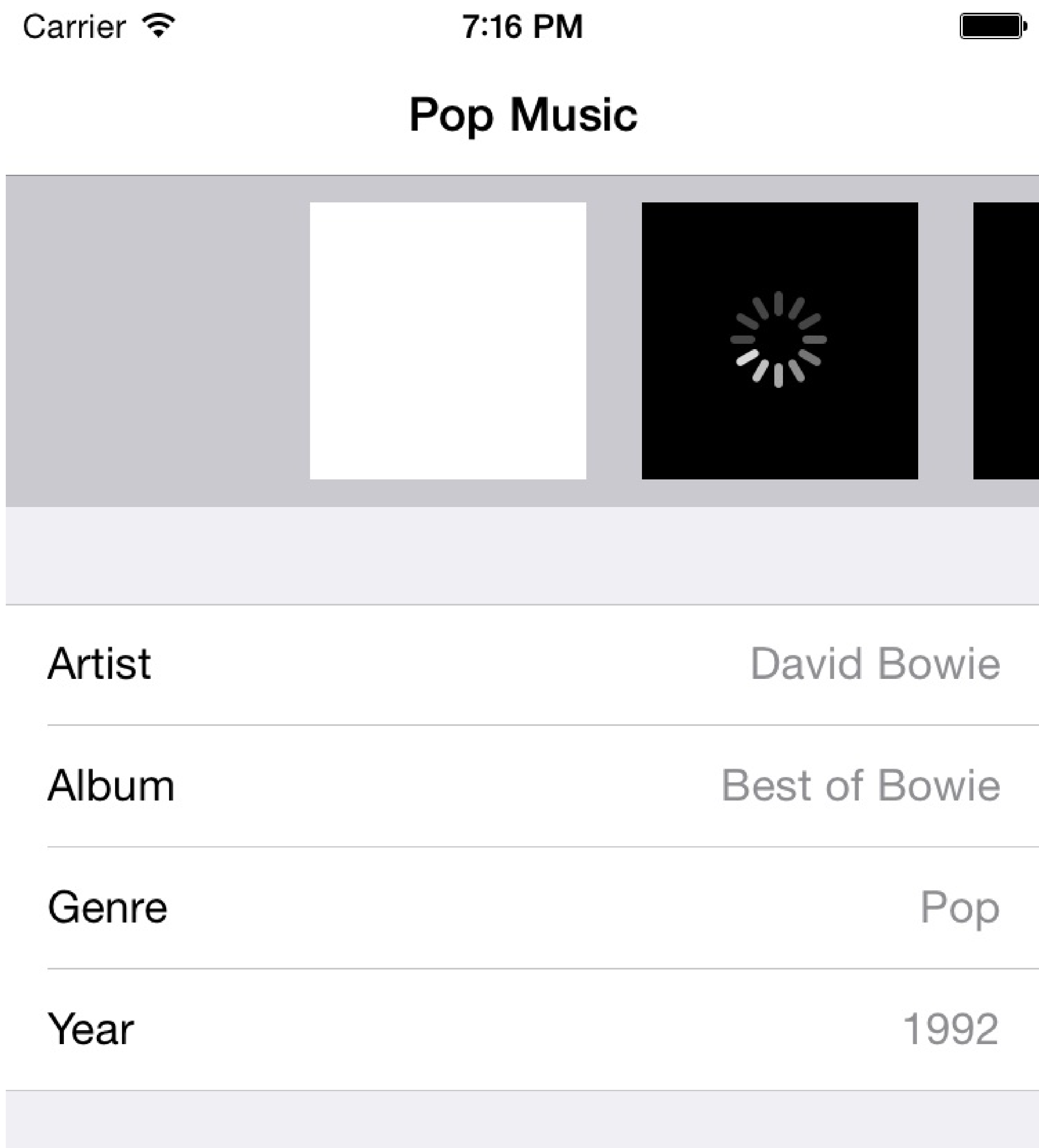
接下来只需要指定委托就可以了，在 `viewDidLoad` 最后加入一下代码：

```
scroller.delegate = self
reloadScroller()
```

因为 `HorizontalScroller` 是在 `StoryBoard` 里初始化的，所以我们需要做的只是指定委托，然后调用 `reloadScroller()` 方法，从而加载所有的子视图并且展示专辑数据。

标注：如果协议里的方法过多，可以考虑把它分解成几个更小的协议。 `UITableViewDelegate` 和 `UITableViewDataSource` 就是很好的例子，它们都是 `UITableView` 的协议。尝试去设计你自己的协议，让每个协议都单独负责一部分功能。

运行一下当前项目，看一下我们的新页面：



等下，滚动视图显示出来了，但是专辑的封面怎么不见了？

啊哈，是的。我们还没完成下载部分的代码，我们需要添加下载图片的方法。因为我们所有的访问都是通过 `LibraryAPI` 实现的，所以很显然我们下一步应该去完善这个类了。不过在这之前，我们还需要考虑一些问题：

- `AlbumView` 不应该直接和 `LibraryAPI` 交互，我们不应该把视图的逻辑和业务逻辑混在一起。
- 同样， `LibraryAPI` 也不应该知道 `AlbumView` 这个类。
- 如果 `AlbumView` 要展示封面， `LibraryAPI` 需要告诉 `AlbumView` 图片

下载完成。

看起来好像很难的样子？别绝望，接下来我们会用观察者模式（`Observer Pattern`）解决这个问题！:]

完成到这一步的Demo：

- 查看源码
- 下载Zip

观察者模式 - Observer

在观察者模式里，一个对象在状态变化的时候会通知另一个对象。参与者并不需要知道其他对象的具体是干什么的 - 这是一种降低耦合度的设计。这个设计模式常用于在某个属性改变的时候通知关注该属性的对象。

常见的使用方法是观察者注册监听，然后再状态改变的时候，所有观察者们都会收到通知。

在 MVC 里，观察者模式意味着需要允许 `Model` 对象和 `View` 对象进行交流，而不能有直接的关联。

`Cocoa` 使用两种方式实现了观察者模式： `Notification` 和 `Key-Value Observing (KVO)` 。

通知 - Notification

不要把这里的通知和推送通知或者本地通知搞混了，这里的通知是基于订阅-发布模型的，即一个对象 (发布者) 向其他对象 (订阅者) 发送消息。发布者永远不需要知道订阅者的任何数据。

Apple 对于通知的使用很频繁，比如当键盘弹出或者收起的时候，系统会分别发送 `UIKeyboardWillShowNotification/UIKeyboardWillHideNotification` 的通知。当你的应用切到后台的时候，又会发送 `UIApplicationDidEnterBackgroundNotification` 的通知。

注意：打开 `UIApplication.swift` 文件，在文件结尾你会看到二十多种系统发送的通知。

如何使用通知

打开 `AlbumView.swift` 然后在 `init` 的最后插入如下代码：

```
NSNotificationCenter.defaultCenter().postNotificationName("BLDownlo
```

这行代码通过 `NSNotificationCenter` 发送了一个通知，通知信息包含了 `UIImageView` 和图片的下载地址。这是下载图像需要的所有数据。

然后在 `LibraryAPI.swift` 的 `init` 方法的 `super.init()` 后面加上如下代码：

```
NSNotificationCenter.defaultCenter().addObserver(self, selector:"do
```

这是等号的另一边：观察者。每当 `AlbumView` 发出一个 `BLDownloadImageNotification` 通知的时候，由于 `LibraryAPI` 已经注册了成为观察者，所以系统会调用 `downloadImage()` 方法。

但是，在实现 `downloadImage()` 之前，我们必须先在 `dealloc` 里取消监听。如果没有取消监听消息，消息会发送给一个已经销毁的对象，导致程序崩溃。

在 `LibaratyAPI.swift` 里加上取消订阅的代码：

```
deinit {
    NSNotificationCenter.defaultCenter().removeObserver(self)
}
```

当对象销毁的时候，把它从所有消息的订阅列表里去除。

这里还要做一件事情：我们最好把图片存储到本地，这样可以避免一次又一次下载相同的封面。

打开 `PersistencyManager.swift` 添加如下代码：

```
func saveImage(image: UIImage, filename: String) {
    let path = NSHomeDirectory().stringByAppendingString("/Document
    let data = UIImagePNGRepresentation(image)
    data.writeToFile(path, atomically: true)
}

func getImage(filename: String) -> UIImage? {
    let path = NSHomeDirectory().stringByAppendingString("/Document
    do {
        let data =  try NSData(contentsOfFile: path, options: .Unca
        return UIImage(data: data)
    } catch _ {
        return nil
    }
}
```

代码很简单直接，下载的图片会存储在 `Documents` 目录下，如果没有检查到缓存文件， `getImage()` 方法则会返回 `nil` 。

然后在 `LibraryAPI.swift` 添加如下代码：

```
func downloadImage(notification: NSNotification) {
    //1
    let userInfo = notification.userInfo as! [String: AnyObject]
    let imageView = userInfo["imageView"] as! UIImageView?
    let coverUrl = userInfo["coverUrl"] as! NSString

    //2
    if let imageViewUnWrapped = imageView {
        imageViewUnWrapped.image = persistencyManager.getImage(cove
        if imageViewUnWrapped.image == nil {
            //3
            dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE
                let downloadedImage = self.httpClient.downloadImage
                //4
                dispatch_sync(dispatch_get_main_queue(), { () -> Vo
                    imageViewUnWrapped.image = downloadedImage
                    self.persistencyManager.saveImage(downloadedIma
                })
            })
        }
    }
}
```

拆解一下上面的代码：

- `downloadImage` 通过通知调用，所以这个方法的参数就是 `NSNotification` 本身。 `UIImageView` 和 `URL` 都可以从其中获取到。
- 如果以前下载过，从 `PersistencyManager` 里获取缓存。
- 如果图片没有缓存，则通过 `HTTPClient` 获取。
- 如果下载完成，展示图片并用 `PersistencyManager` 存储到本地。

再回顾一下，我们使用外观模式隐藏了下载图片的复杂程度。通知的发送者并不在乎图片是如何从网上下载到本地的。

如果你是 `Xcode 7` 和 iOS9 那么运行项目，程序会崩溃同时看到控制台有如下输出：

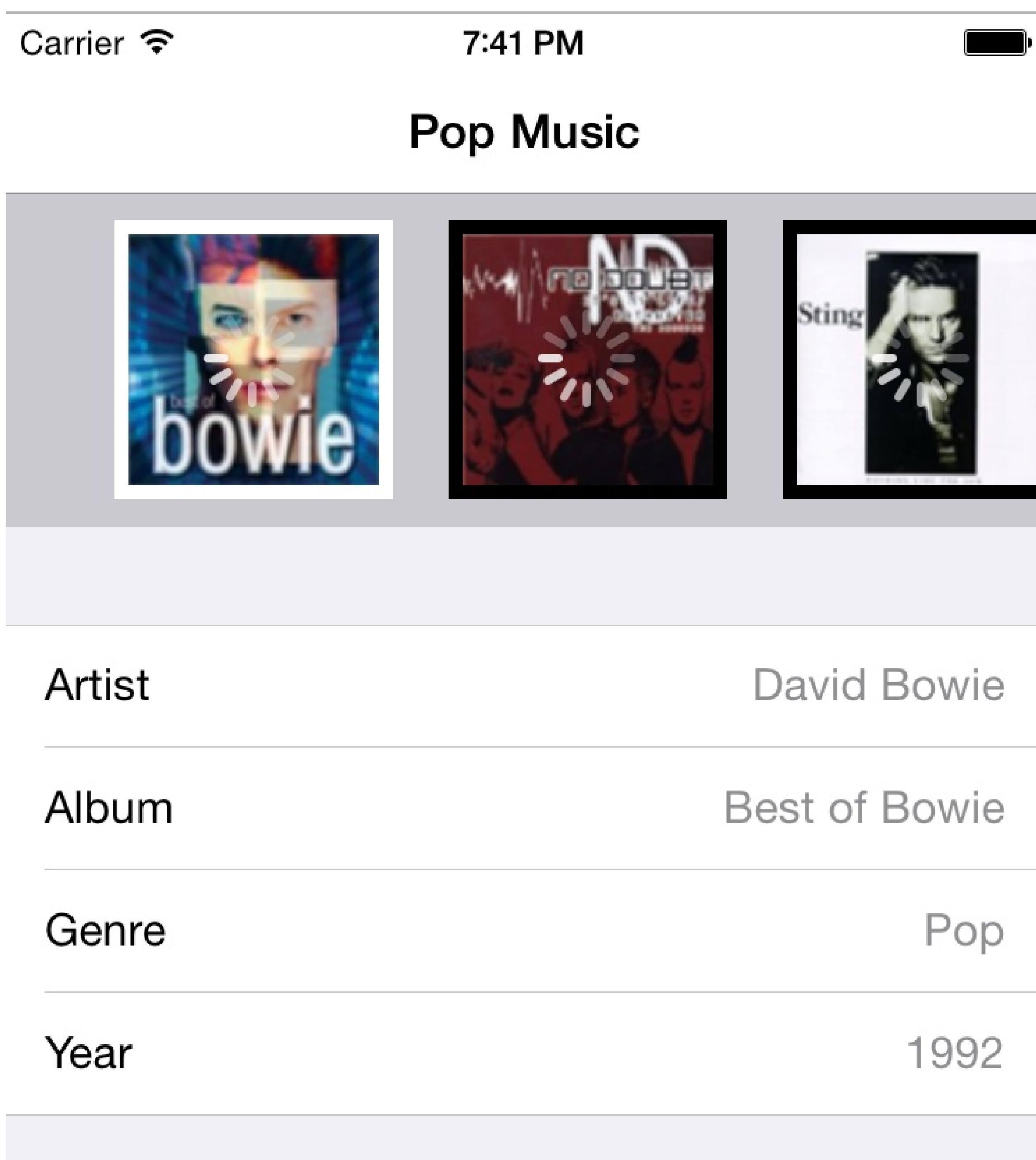
```
App Transport Security has blocked a cleartext HTTP (http://) resou
```

解决方法是如下（参考：解决iOS9下blocked cleartext HTTP）

修改项目的 Info.plist 文件，增加以下内容：

```
<key>NSAppTransportSecurity</key>
<dict>
    <key>NSAllowsArbitraryLoads</key>
    <true/>
</dict>
```

现在运行一下项目，可以看到专辑封面已经显示出来了：



关了应用再重新运行，注意这次没有任何延时就显示了所有的图片，因为我们已经有了本地缓存。我们甚至可以在没有网络的情况下正常使用我们的应用。不过出了问题：这个用来提示加载网络请求的小菊花怎么一直在显示！

我们在下载图片的时候开启了这个白色小菊花，但是在图片下载完毕的时候我们并没有停掉它。我们可以在每次下载成功的时候发送一个通知，但是我们不这样做，这次我们来用用另一个观察者模式： `KVO` 。

完成到这一步的Demo：

- 查看源码

- 下载Zip

键值观察 - KVO

在 KVO 里，对象可以注册监听任何属性的变化，不管它是否持有。如果感兴趣的话，可以读一读苹果 KVO 编程指南。

如何使用 KVO

正如前面所提及的， 对象可以关注任何属性的变化。在我们的例子里，我们可以用 KVO 关注 `UIImageView` 的 `image` 属性变化。

打开 `AlbumView.swift` 文件，找到 `init(frame:albumCover:)` 方法，在把 `coverImage` 添加到 `subView` 的代码后面添加如下代码：

```
coverImage.addObserver(self, forKeyPath: "image", options: NSKeyVal
```

这行代码把 `self` (也就是当前类) 添加到了 `coverImage` 的 `image` 属性的观察者里。

在销毁的时候，我们也需要取消观察。还是在 `AlbumView.swift` 文件里，添加如下代码：

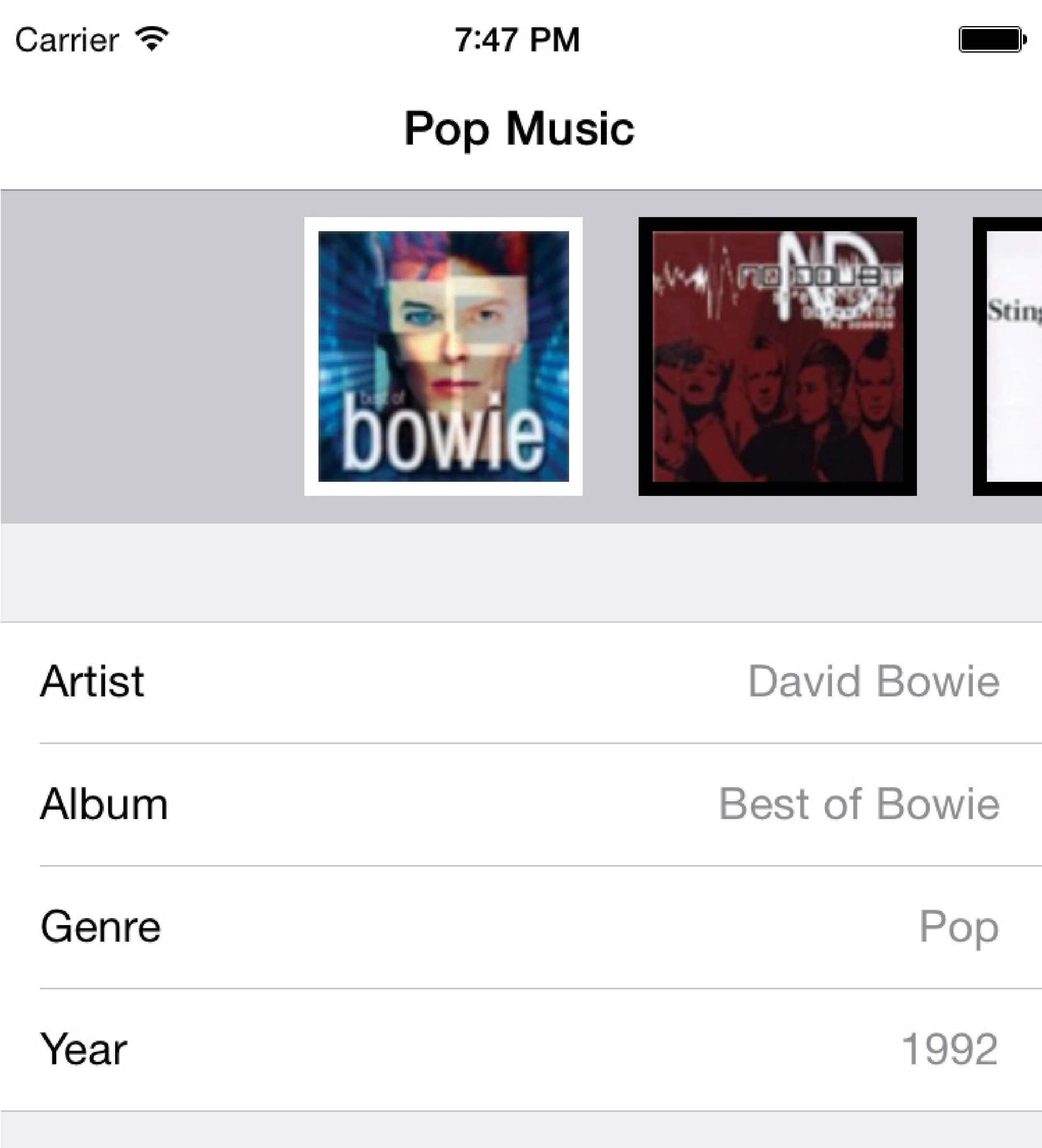
```
deinit {
    coverImage.removeObserver(self, forKeyPath: "image")
}
```

最终添加如下方法：

```
override func observeValueForKeyPath(keyPath: String?, ofObject obj
    if keyPath == "image" {
        indicator.stopAnimating()
    }
}
```

必须在所有的观察者里实现上面的代码。在检测到属性变化的时候，系统会自动调用这个方法。在上面的代码里，我们在图片加载完成的时候把那个提示加载的小菊花去掉了。

再次运行项目，你会发现一切正常了：



注意：一定要记得移除观察者，否则如果对象已经销毁了还给它发送消息会导致应用崩溃。

此时你可以把玩一下当前的应用然后再关掉它，你会发现你的应用的状态并没有存储下来。最后看见的专辑并不会再下次打开应用的时候出现。

为了解决这个问题，我们可以使用下一种模式：备忘录模式。

完成到这一步的Demo：

- 查看源码
- 下载Zip

备忘录模式 - Memento

备忘录模式捕捉并且具象化一个对象的内在状态。换句话说，它把你的对象存在了某个地方，然后在以后的某个时间再把它恢复出来，而不会打破它本身的封装性，私有数据依旧是私有数据。

如何使用备忘录模式

在 `ViewController.swift` 里加上下面两个方法：

```
//MARK: Memento Pattern
func saveCurrentState() {
    // When the user leaves the app and then comes back again, he w
    // he left it. In order to do this we need to save the currentl
    // Since it's only one piece of information we can use NSUserDe
    NSUserDefaults.standardUserDefaults().setInteger(currentAlbumIn
}

func loadPreviousState() {
    currentAlbumIndex = NSUserDefaults.standardUserDefaults().integ
    showDataForAlbum(currentAlbumIndex)
}
```

`saveCurrentState` 把当前相册的索引值存到 `NSUserDefaults` 里。 `NSUserDefaults` 是 iOS 提供的一个标准存储方案，用于保存应用的配置信息和数据。

`loadPreviousState` 方法加载上次存储的索引值。这并不是备忘录模式的完整实现，但是已经离目标不远了。

接下来在 `viewDidLoad` 的 `scroller.delegate = self` 前面调用：

```
loadPreviousState()
```

这样在刚初始化的时候就加载了上次存储的状态。但是什么时候存储当前状态呢？这个时候我们可以用通知来做。在应用进入到后台的时候， iOS 会发送一个 `UIApplicationDidEnterBackgroundNotification` 的通知，我们可以在这个通知里调用 `saveCurrentState` 这个方法。是不是很方便？

在 `viewDidLoad` 的最后加上如下代码：

```
NSNotificationCenter.defaultCenter().addObserver(self, selector:"sa
```

现在，当应用即将进入后台的时候，`ViewController` 会调用 `saveCurrentState` 方法自动存储当前状态。

当然也别忘了取消监听通知，添加如下代码：

```
deinit {
    NSNotificationCenter.defaultCenter().removeObserver(self)
}
```

这样就确保在 `ViewController` 销毁的时候取消监听通知。

这时再运行程序，随意移到某个专辑上，然后按下 Home 键把应用切换到后台，再在 Xcode 上把 App 关闭。重新启动，会看见上次记录的专辑已经存了下来并成功还原了：

Carrier 8:00 PM

Pop Music

Artist	U2
Album	Staring at the Sun
Genre	Pop
Year	2000

看起来专辑数据好像是对了，但是上面的滚动条似乎出了问题，没有居中啊！

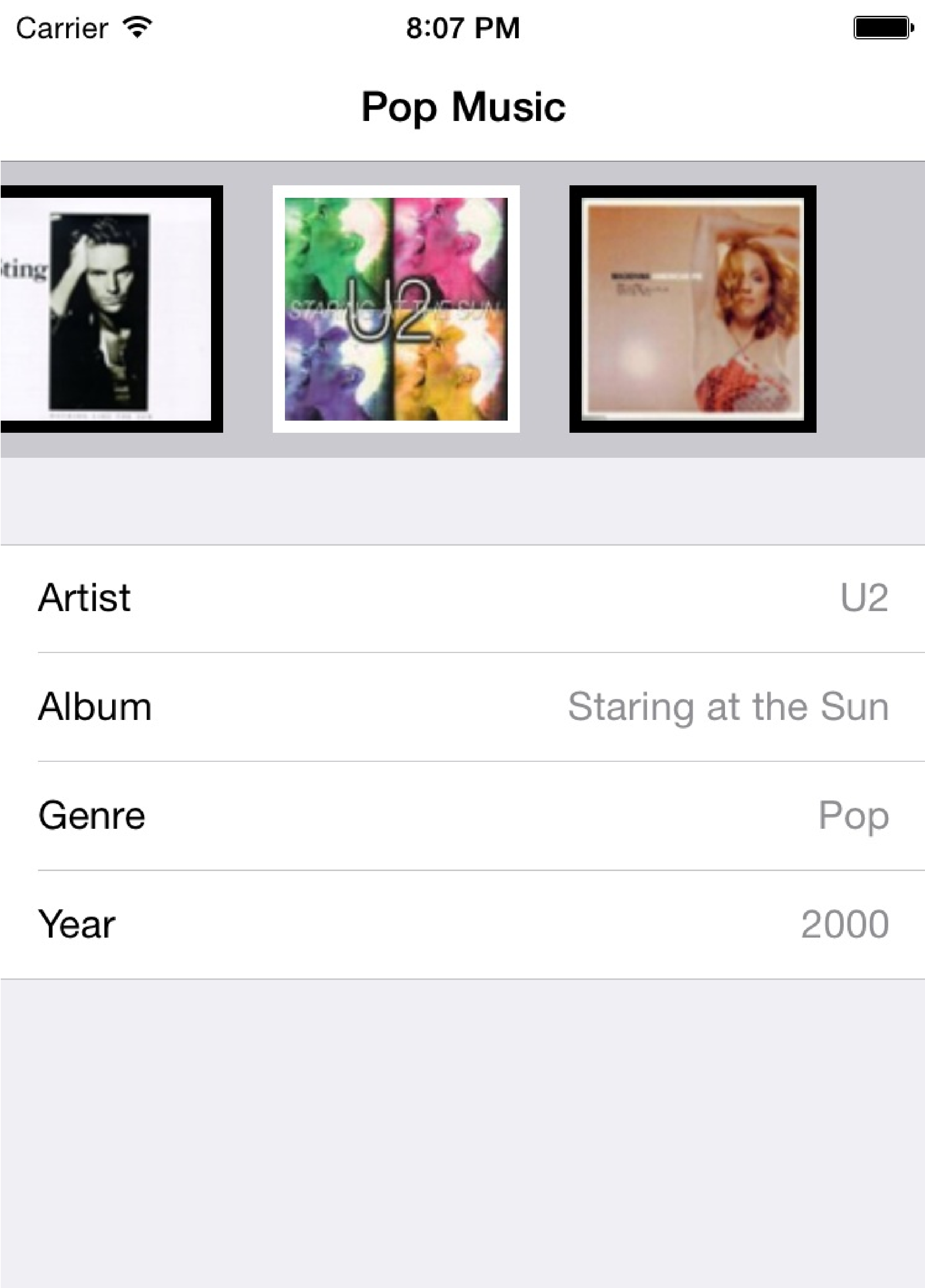
这时 `initialViewIndex` 方法就派上用场了。由于在委托里 (也就是 `ViewController`) 还没实现这个方法，所以初始化的结果总是第一张专辑。

为了修复这个问题，我们可以在 `ViewController.swift` 里添加如下代码：

```
func initialViewIndex(scroller: HorizontalScroller) -> Int {
    return currentAlbumIndex
}
```

现在 `HorizontalScroller` 可以根据 `currentAlbumIndex` 自动滑到相应的索引位置了。

再次重复上次的步骤，切到后台，关闭应用，重启，一切顺利：



回头看看 `PersistencyManager` 的 `init` 方法，你会发现专辑数据是我们硬编码写进去的，而且每次创建 `PersistencyManager` 的时候都会再创建一次专辑数据。而实际上一个比较好的方案是只创建一次，然后把专辑数据存到本地文件里。

我们如何把专辑数据存到文件里呢？

一种方案是遍历 Album 的属性然后把它们写到一个 `plist` 文件里，然后如果需要的时候再重新创建 `Album` 对象。这并不是最好的选择，因为数据和属性不同，你的代码也就要相应的产生变化。举个例子，如果我们以后想添加 `Movie` 对象，它有着完全不同的属性，那么存储和读取数据又需要重写新的代码。

况且你也无法存储这些对象的私有属性，因为其他类是没有访问权限的。这也就是为什么 Apple 提供了 归档 的机制。

完成到这一步的Demo：

- 查看源码
- 下载Zip

归档 - Archiving

苹果通过归档的方法来实现备忘录模式。它把对象转化成了流然后在不暴露内部属性的情况下存储数据。你可以读一读 《iOS 6 by Tutorials》 这本书的第 16 章，或者看下苹果的归档和序列化文档。

如何使用归档

首先，我们需要让 `Album` 实现 `NSCoding` 协议，声明这个类是可被归档的。打开 `Album.swift` 在 `class` 那行后面加上 `NSCoding` ：

```
class Album: NSObject, NSCoding {
```

然后添加如下的两个方法：

```
required init(coder decoder: NSCoder) {
    super.init()
    self.title = decoder.decodeObjectForKey("title") as! String
    self.artist = decoder.decodeObjectForKey("artist")as! String
    self.genre = decoder.decodeObjectForKey("genre") as! String?
    self.coverUrl = decoder.decodeObjectForKey("cover_url")as! Stri
    self.year = decoder.decodeObjectForKey("year") as! String
}

func encodeWithCoder(aCoder: NSCoder) {
    aCoder.encodeObject(title, forKey: "title")
    aCoder.encodeObject(artist, forKey: "artist")
    aCoder.encodeObject(genre, forKey: "genre")
    aCoder.encodeObject(coverUrl, forKey: "cover_url")
    aCoder.encodeObject(year, forKey: "year")
}
```

`encodeWithCoder` 方法是 `NSCoding` 的一部分，在被归档的时候调用。相对的， `init(coder:)` 方法则是用来解档的。很简单，很强大。

现在 `Album` 对象可以被归档了，添加一些代码来存储和加载 `Album` 数据。

在 `PersistencyManager.swift` 里添加如下代码：

```
func saveAlbums() {
    let filename = NSHomeDirectory().stringByAppendingString("/Docu
    let data = NSKeyedArchiver.archivedDataWithRootObject(albums)
    data.writeToFile(filename, atomically: true)
}
```

这个方法可以用来存储专辑。 `NSKeyedArchiver` 把专辑数组归档到了 `albums.bin` 这个文件里。

当我们归档一个包含子对象的对象时，系统会自动递归的归档子对象，然后是子对象的子对象，这样一层层递归下去。在我们的例子里，我们归档的是 `albums` 因为 `Array` 和 `Album` 都是实现 `NSCopying` 接口的，所以数组里的对象都可以自动归档。

用下面的代码取代 `PersistencyManager` 中的 `init` 方法：

```
override init() {
    super.init()
    if let data = NSData(contentsOfFile: NSHomeDirectory().stringBy
        let unarchiveAlbums = NSKeyedUnarchiver.unarchiveObjectWith
        if let unwrappedAlbum : [Album]  = unarchiveAlbums {
            albums = unwrappedAlbum
        }
    } else {
        createPlaceholderAlbum()
    }
}

func createPlaceholderAlbum() {
    //Dummy list of albums
    let album1 = Album(title: "Best of Bowie",
             artist: "David Bowie",
             genre: "Pop",
             coverUrl: "http://www.coversproject.com/static/thumbs/
             year: "1992")
```

```
    let album2 = Album(title: "It's My Life",
           artist: "No Doubt",
           genre: "Pop",
           coverUrl: "http://www.coversproject.com/static/thumbs/al
           year: "2003")

    let album3 = Album(title: "Nothing Like The Sun",
               artist: "Sting",
           genre: "Pop",
           coverUrl: "http://www.coversproject.com/static/thumbs/al
           year: "1999")

    let album4 = Album(title: "Staring at the Sun",
           artist: "U2",
           genre: "Pop",
           coverUrl: "http://www.coversproject.com/static/thumbs/al
           year: "2000")

    let album5 = Album(title: "American Pie",
           artist: "Madonna",
           genre: "Pop",
           coverUrl: "http://www.coversproject.com/static/thumbs/al
           year: "2000")
    albums = [album1, album2, album3, album4, album5]
    saveAlbums()
}
```

我们把创建专辑数据的方法放到了 `createPlaceholderAlbum` 里，这样代码可读性更高。在新的代码里，如果存在归档文件， `NSKeyedUnarchiver` 从归档文件加载数据；否则就创建归档文件，这样下次程序启动的时候可以读取本地文件加载数据。

我们还想在每次程序进入后台的时候存储专辑数据。看起来现在这个功能并不是必须的，但是如果以后我们加了编辑功能，这样做还是很有必要的，那时我们肯定希望确保新的数据会同步到本地的归档文件。

因为我们的程序通过 `LibraryAPI` 来访问所有服务，所以我们需要通过 `LibraryAPI` 来通知 `PersistencyManager` 存储专辑数据。

在 `LibraryAPI` 里添加存储专辑数据的方法：

```
func saveAlbums() {
    persistencyManager.saveAlbums()
}
```

这个方法很简单，就是把 `LibraryAPI` 的 `saveAlbums` 方法传递给了 `persistencyManager` 的 `saveAlbums` 方法。

然后在 `ViewController.swift` 的 `saveCurrentState` 方法的最后加上：

```
LibraryAPI.sharedInstance.saveAlbums()
```

在 `ViewController` 需要存储状态的时候，上面的代码通过 `LibraryAPI` 归档当前的专辑数据。

运行一下程序，检查一下没有编译错误。

不幸的是似乎没什么简单的方法来检查归档是否正确完成。你可以检查一下 `Documents` 目录，看下是否存在归档文件。如果要查看其他数据变化的话，还需要添加编辑专辑数据的功能。

不过和编辑数据相比，似乎加个删除专辑的功能更好一点，如果不想要这张专辑直接删除即可。再进一步，万一误删了话，是不是还可以再加个撤销按钮？

完成到这一步的Demo：

- 查看源码
- 下载Zip

最后的润色

现在我们将添加最后一个功能：允许用户删除专辑，以及撤销上次的删除操作。

在 `ViewController` 里添加如下属性：

```
// 为了实现撤销功能，我们用数组作为一个栈来 push 和 pop 用户的操作
var undoStack: [(Album, Int)] = []
```

然后在 `viewDidLoad` 的 `reloadScroller()` 后面添加如下代码：

```
let undoButton = UIBarButtonItem(barButtonSystemItem: .Undo, target
undoButton.enabled = false;
let space = UIBarButtonItem(barButtonSystemItem: .FlexibleSpace, ta
let trashButton = UIBarButtonItem(barButtonSystemItem: .Trash, targ
let toolbarButtonItems = [undoButton, space, trashButton]
toolbar.setItems(toolbarButtonItems, animated: true)
```

上面的代码创建了一个 `toolbar` ，上面有两个按钮，在 `undoStack` 为空的情况下， `undo` 的按钮是不可用的。注意 `toolbar` 已经在 `storyboard` 里了，我们需要做的只是配置上面的按钮。

我们需要在 `ViewController.swift` 里添加三个方法，用来处理专辑的编辑事件：增加，删除，撤销。

先写添加的方法：

```
func addAlbumAtIndex(album: Album,index: Int) {
    LibraryAPI.sharedInstance.addAlbum(album, index: index)
    currentAlbumIndex = index
    reloadScroller()
}
```

做了三件事：添加专辑，设为当前的索引，重新加载滚动条。

接下来是删除方法：

```
func deleteAlbum() {
    //1
    let deletedAlbum : Album = allAlbums[currentAlbumIndex]
    //2
    let undoAction = (deletedAlbum, currentAlbumIndex)
    undoStack.insert(undoAction, atIndex: 0)
    //3
    LibraryAPI.sharedInstance.deleteAlbum(currentAlbumIndex)
    reloadScroller()
    //4
    let barButtonItems = toolbar.items! as [UIBarButtonItem]
    let undoButton : UIBarButtonItem = barButtonItems[0]
    undoButton.enabled = true
    //5
    if (allAlbums.count == 0) {
        let trashButton : UIBarButtonItem = barButtonItems[2]
        trashButton.enabled = false
    }
}
```

挨个看一下各个部分：

- 获取要删除的专辑。
- 创建一个 `undoAction` 对象，用元组存储 `Album` 对象和它的索引值。然后把这个元组加到了栈里。
- 使用 `LibraryAPI` 删除专辑数据，然后重新加载滚动条。
- 既然撤销栈里已经有了数据，那么我们需要设置撤销按钮为可用。
- 检查一下是不是还剩专辑，如果没有专辑了那就设置删除按钮为不可用。

最后添加撤销按钮：

```
func undoAction() {
    let barButtonItems = toolbar.items! as [UIBarButtonItem]
    //1
    if undoStack.count > 0 {
        let (deletedAlbum, index) = undoStack.removeAtIndex(0)
        addAlbumAtIndex(deletedAlbum, index: index)
    }
    //2
    if undoStack.count == 0 {
        let undoButton : UIBarButtonItem = barButtonItems[0]
        undoButton.enabled = false
    }
    //3
    let trashButton : UIBarButtonItem = barButtonItems[2]
    trashButton.enabled = true
}
```

照着备注的三个步骤再看一下撤销方法里的代码：

- 首先从栈里 `pop` 出一个对象，这个对象就是我们当初塞进去的元祖，存有删除的 `Album` 对象和它的索引位置。然后我们把取出来的对象放回了数据源里。
- 因为我们从栈里删除了一个对象，所以需要检查一下看看栈是不是空了。如果空了那就设置撤销按钮不可用。
- 既然我们已经撤消了一个专辑，那删除按钮肯定是可用的。所以把它设置为 `enabled` 。

这时再运行应用，试试删除和插销功能，似乎一切正常了：

Carrier 9:51 AM

Pop Music

Sting

U2 STARING AT THE SUN

Artist	David Bowie
Album	Best of Bowie
Genre	Pop
Year	1992

Undo

我们也可以趁机测试一下，看看是否及时存储了专辑数据的变化。比如删除一个专辑，然后切到后台，强关应用，再重新开启，看看是不是删除操作成功保存了。

如果想要恢复所有数据，删除应用然后重新安装即可。

完成到这一步的Demo：

- 查看源码
- 下载Zip

小结

最终项目的源代码可以在 BlueLibrarySwift-Final 下载。

通过这两篇设计模式的学习，我们接触到了一些基础的设计模式和概念： `Singleton` 、 `MVC` 、 `Delegation` 、 `Protocols` 、 `Facade` 、 `Observer` 、 `Memento` 。

这篇文章的目的，并不是推崇每行代码都要用设计模式，而是希望大家在考虑一些问题的时候，可以参考设计模式提出一些合理的解决方案，尤其是应用开发的起始阶段，思考和设计尤为重要。

如果想继续深入学习设计模式，推荐设计模式的经典书籍：Design Patterns: Elements of Reusable Object-Oriented Software。

如果想看更多的设计模式相关的代码，推荐这个神奇的项目： Swift 实现的种种设计模式。

接下来你可以看看这篇：Swift 设计模式中级指南，学习更多的设计模式。

玩的开心。 :]